[19] 中华人民共和国国家知识产权局

[51] Int. Cl[7]
G06F

[12] 发明专利申请公开说明书

[21] 申请号 200410029453.3

[43] 公开日 2005 年 9 月 21 日

[11] 公开号 CN 16706

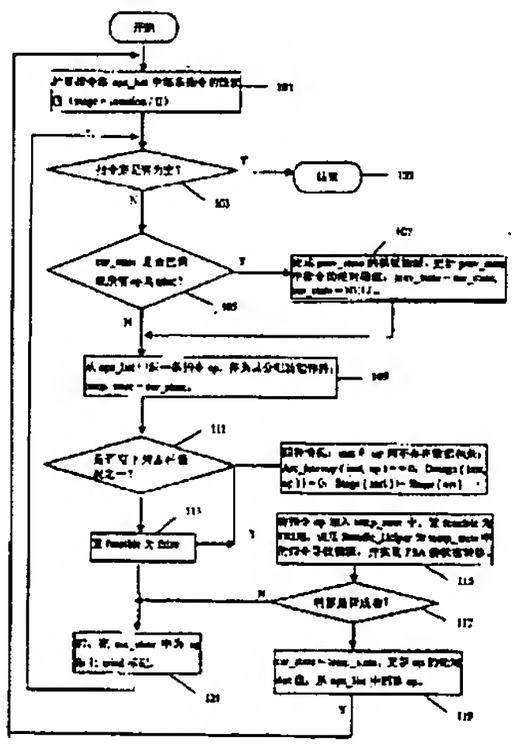
[22] 申请日 2004.3.19
[21] 申请号 200410029453.3
[71] 申请人 中国科学院计算技术研究所
地址 100080 北京市海淀区中关村科学院南路 6 号
[72] 发明人 文严治 连瑞琦 刘章林 吴承勇 张兆庆
[74] 专利代理机构 北京泛华伟业知识产权代理公司
代理人 王凤华

权利要求书 2 页 说明书 18 页 附图 3 页

[54] 发明名称 一种支持有向有环图的微调度方法

[57] 摘要

本发明涉及一种支持有向有环图的微调度方法，在运用重排(Reorder)技术和协调(Negotiate)技术编排模调度认为能在同一 cycle 中发射的指令集合的时候，除了考虑指令间的依赖关系以外，还要同时考虑指令间弧上的延迟值和指令所在的级数，实现对“回边”的支持；避免软件流水模调度中出现的分拆问题(Split issue)，减小了出现指令 cache 访问不命中(I－Cache miss)的可能性，提高了并行编译效率，从而进一步提高了编译优化性能。



1、一种支持有向有环图的微调度方法，其特征在于包括以下步骤：

a) 计算指令集中每条指令的级数值；

b) 判断指令集是否为空？如果是，执行步骤 1)；如果否，执行下一步；

c) 判断机器当前状态空间是否已满或所有指令均被选过？若是，执行下一步，若否，执行步骤 e)；

d) 完成前一周期的机器状态空间的模板指派，更新前一周期的机器状态空间中指令的绝对槽值，把当前周期的机器状态空间赋给前一周期的状态空间，把当前机器的状态空间置空；

e) 从指令集中选一指令，为其分配功能部件，把当前机器的状态空间值赋给当前周期的测试空间；

f) 根据数据依赖图，检查当前机器的状态空间的每一条指令与步骤 e)中选取指令的相关性，即判断是否有下列四种情况之一来判断相关性，如有，则执行步骤 h)，如否，则执行下一步；四种情况为：当前周期的机器状态空间中的任一条指令和所选指令间不存在数据相关；数据依赖图中任一指令与所选指令的弧上延迟值为 0；数据依赖图中任一指令到所选指令的弧上的循环迭代数差值不为 0；模调度中任一指令所在的级数不为所选指令所在的级数；

g) 置 feasible 为 false，然后执行步骤 k)，其中，feasible 为判断是否存在周期内的依赖关系的逻辑变量；

h) 将指令 op 加入当前周期的测试空间中，置 feasible 为 TRUE，调用模板匹配函数为当前周期的测试空间状态中的指令寻找模板，并实现有限状态自动机的状态转移；

i) 判断有限状态自动机的状态转移是否成功？如果是，执行下一步，如果否，执行步骤 l)；

j) 测试成功，把当前周期的测试空间的值赋给当前周期的机器状态空间，更新所选指令的绝对槽值，从指令集中删除所选指令，然后执行步骤 b)；

k) 在当前周期的机器状态空间中为所选指令做上已选标记，然后执行步骤 c)；

l) 微调度结束。

2、如权利要求 1 所述的一种支持有向有环图的微调度方法，其特征在于，所述指令集为模调度认为放在同一 cycle 中发送的指令集。

3、如权利要求 1 所述的一种支持有向有环图的微调度方法，其特征在于步骤 a)中所述每条指令的级数值为所述指令在扁平调度中的位置除以启动间距。

4、如权利要求 1 所述的一种支持有向有环图的微调度方法，其特征在于，所述绝对槽值表示所述方法结束后指令的绝对槽值。

一种支持有向有环图的微调度方法

技术领域

本发明涉及指令级并行性编译、微调度技术领域，特别是涉及一种支持带“回边”的“有向有环图”的微调度方法，及相应编译优化技术。

背景技术

指令调度是机器相关优化的重要阶段。成功的指令调度需要满足数据相关，控制相关和结构相关及其它约束条件，通过重排指令顺序提高资源利用率和指令并行度。而其中结构相关，是指如果指令并行执行，可能发生的资源冲突。要解决这类相关需要访问占用资源状态，访问机器状态，获取指令间延迟等等，所以需要频繁的访问机器模型【Muchnick. Advance Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997】。

另外，有的机器对指令发送顺序也有特殊要求，如 Intel 公司基于 IA-64 体系结构的第一代、第二代处理器—安腾（Itanium）和麦金利（McKinley 即 itanium2）。它们都需要机器以特殊的模板，例如 MII，发射指令。其中 MII 表示第一条发射 Memory 指令，而后两条必须是 Integer 指令。为此指令调度需要考虑更加复杂的因素，这对代码移植也是十分不利的。

鉴于性能和移植性的双重需求，我们将指令调度考虑结构相关和其它限制条件的部分单独拿出来，使之成为一个新模块，称为*微调度*【Dong-Yuan Chen, Lixia Liu, Chen Fu, Shuxin Yang, Chengyong Wu, Roy Ju. Efficient Resource Management during Instruction Scheduling for the EPIC Architecture. Parallel Architectures and Compilation Techniques. 2003, 9 : 36∽45】。它负责封装目标机微体系结构的具体细节。一方面为其它机器相关的优化提供访问机器参数以及机器状态的接口，使编译器在一定程度上与机器无关，能够快速适应目标机硬件的修改或换代。另一方面，也提高了指令调度的灵活性与可移植性。

首先，微调度的工作涉及范围小，对于机器状态转换，我们只需关心当前周期的状态变化即可，加上特殊限制，我们最多需要考虑前面几个的周期状态。那样也就是几个周期内的调度范围。其次，我们必须考虑为指令分配功能部件，考虑指令的重排，这就要求在周期内作一种类似调度一样的工作。

在满足结构相关的过程中，微调度部分还必须得知当前机器占用资源的状态，当选择了合适的指令发射之后，状态要随之相应的变化。这种机器状态变化的过程和有限状态自动机状态转换类似。

通常，指令调度解决结构相关有两种方法，向后检测和向前检测。向后方法是记录当前周期已发射的指令，对于每个待发射的指令，判断

是否和当前周期已发射的指令发生资源冲突或者违背其它限制。如果存在冲突或者违背，则不允许该指令在当前周期发射，否则允许发射。而前向检测方法是对于发射每条指令都记录了发射后的机器状态，对于新的候选指令，从当前机器状态判断是否能够到达下一个合法状态。如果不能则不允许指令发射，否则反之。两种方法一种向后看，看过去发射的指令，另一种向前看判断下一个状态是否合法【T. Muller. Employing finite automata for resource scheduling. In the 26th Annual International Symposium on Microarchitecture, Austin, Dec 1993: 12∽20】。后向方法需要对于每条指令有复杂的比较和判断，速度慢且算法复杂，但它能处理特殊情况，并且空间消耗小。而前向方法通过状态转移，速度快，算法简单，但是要想模拟所有状态，空间消耗大，而且难以处理特殊情况。微调度结合了这两种方法各自的优点，实现了一种处理器结构相关的混合方法。

目前微调度的思想已经在ORC中得到了成功的应用【Open Research Compiler (ORC). http://ipf-orc.sourceforge.net. 2002.】。但它只是实现了对“有向无环图”的调度，还不支持“有向有环图”的情况，故还不能应用于软件流水中，实现对带“回边”指令的调度。另一方面，ORC 中软件流水（SWP : Software Pipeline）模块里编排指令束（bundling）的原有部分，其算法简单，没有微调度【Richard A. Huff. Lifetime-sensitive modulo scheduling. ACM SIGPLAN Notices, 1993,28(6) : 258∽267】，除了不够灵活以外，还丧失许多潜在的提高

优化性能的机会。

作为一种循环调度方法，软件流水（SWP）可以有效的开发程序中隐藏的指令级并行性(Instruction Level Parallelism，简称 ILP)。它通过重迭不同循环体的执行来提高速度。在一个循环体尚未执行完毕之前，可以启动下一个循环体，二者之间的时间差距称为启动间距(Initiation interval，简称 II)。目前软件流水调度有两大主要技术：Move-then-scheduling 技术（也即代码移动技术）和 Schedule-then-move 技术。前者是跨越循环回边(back-edge)一条一条的移动指令。后者则是直接从零开始形成最终的调度。Schedule-then-move 家族又有两大成员:Unroll based scheduling 和 Modulo Scheduling。前者是既做循环展开(unroll)又做指令调度，后者只对循环中的一个 iteration 进行调度，使得每次经过相同的时间间隔重复同样的调度，不会有资源冲突和相关冲突。模调度是目前实现软件流水最受欢迎的方法【Josep M. Codina, Josep Llosa, Antonio González. A Comparative Study of Modulo Scheduling Techniques. Proceedings of the 16th international conference on Supercomputing. 2002,7 : 97∽106】。ORC 中采用了 Richard A. Huff 提出的 Slack Modulo Scheduling【Richard A. Huff. Lifetime-sensitive modulo scheduling. ACM SIGPLAN Notices, 1993,28(6) : 258∽267】。模调度之后紧接的就是编排指令束（bundling）。Itanium 的架构中规定一个周期内能够发射两个指令束

（bundle），每个指令束有三条指令，它们的发送必须要依据一定的模板（template）。Itanium 中指定了 16 种指令模板（MII, MI_I, MLX, MMI, M_MI, MFI, MMF, MIB, MBB, BBB, MMB, MFB, RESERVED_A, RESERVED_D, RESERVED_3, RESERVED_F）。我们把 template 的三个指令位置分别为三个空槽(slot0, slot1, slot2)。ORC 中 SWP 部分的 bundling 源自于 Pro64。其基本思想如下：

```
SWP_Pack_A_Bundle( ops_list )     // ops_list 是模调度认为可以在一个 cycle 中放下的指令集合
{
  while ( !ops_list.empty() ) {                          // ops_list 不为空
    for (each slot in current bundle) {                  // 对当前指令束的每一个 slot
      for (every slot_kind on each slot) {               // 对每个 slot 可能放置的指令类型
        for ( each op in ops_list )                      // 对 ops_list 中的每一个 op
          if  ( !SWP_Violates_Dependencies(ops_list.first_op(), op) ) {
            // op 和它之前未放置的那些 op 没有依赖关系
            SWP_Bundle_Next_In_Group(op, slot, slot_kind ); // 放置该 slot_kind 类型的
                                                            // op 到 slot
            ops_list.erase(op);                          // 删除已放置的 op
          }
      } // end for (every slot_kind…)
    } // end for (each slot…)
  } // while
}
```

算法中 ops_list 是模调度认为可以在一个时钟周期（cycle）中放下的指令集合。其中对每个指令槽（slot）找可能放置的指令类型时，仅仅是根据一种线性的优先级排序。比如，在为一个 bundle 的 slot1 选

择可放置的指令类型时，是按照F > I > M > B的参考顺序来指定指令槽类型（slot_kind）。并且按照算法，一旦前面的slot已经放好指令后，后面的指令不会将其弹出，即使后面的指令找不到合适的template来放置时也如此。而事实上，很多情况下弹出已放置的指令恰恰可以发掘潜在的优化的机会。例如：adds, shr_i.u, add, lfetch, xor, ld8_i六条指令（分别是M/I, I, M/I, M, M/I, M类型），本来可以用MII, MMI(ld8_i, shr_i.u,add ; lfetch, xor, adds)在一个cycle中放下，但SWP的bundling中的线性优先级算法却把它排成了MII, MMF+MFB(adds, shr_i.u, add ; lfetch,xor,nop.f + ld8_I, nop.f, nop.b)也即用了两个cycle才放下。也即模调度认为可以在一个cycle中放下，而事实上没有放下。我们称这种情况为分拆问题（Split issue）【Intel Corp. Itanium Processor Microarchitecture Reference. http://developer.intel.com/design/ itanium/arch spec.htm, 2000】。

另一方面，该算法不仅增加了许多空指令（NOP），增加了指令cache访问不命中（I - Cache miss）的可能性；而且填NOP指令时，优先选用的是nop.f而不是nop.i，这也增加了执行时出现停顿(stall)的可能性【Intel Corp. Intel Itanium2 Processor Reference Manual. 2002,7 : 53∽54】。当这些指令集所在基本块（BB: Basic Block）被执行的频率非常高，并且这些指令集由于SWP做了循环展开（Loop unrolling）而在该基本块中多次重现时，就会显著影响编译优化的性能。也正是这个

例子，直接导致了 SPEC2000 中的 bzip2(ISET=ref)在 ORC peak 选项下编译后，运行时间下降了约 2%。

综上所述，在现有技术软件流水的模调度中，会出现分拆问题，也增加了指令 cache 访问不命中的可能性，降低了编译效率，从而降低了编译优化的性能。

发明内容

本发明要解决的技术问题是提供一种支持有向有环图的微调度方法，避免软件流水模调度中出现的分拆问题(Split issue)，减小了出现指令 cache 访问不命中（I-Cache miss）的可能性，提高了并行编译效率，从而进一步提高了编译优化性能。

为了解决上述技术问题，本发明提供一种支持有向有环图的微调度方法，包括以下步骤：

一种支持有向有环图的微调度方法，其特征在于包括以下步骤：

a) 计算指令集中每条指令的级数值；

b) 判断指令集是否为空？如果是，执行步骤 1)；如果否，执行下一步；

c) 判断机器当前状态空间是否已满或所有指令均被选过？若是，执行下一步，若否，执行步骤 e)；

d) 完成前一周期的机器状态空间的模板指派，更新前一周期的机器状态空间中指令的绝对槽值，把当前周期的机器状态空间赋给前

一周期的状态空间，把当前机器的状态空间置空；

e) 从指令集中选一指令，为其分配功能部件，把当前机器的状态空间赋给当前周期的测试空间；

f) 根据数据依赖图，检查当前机器的状态空间中每一条指令与步骤e)中选取指令的相关性，即判断是否有下列四种情况之一来判断相关性，如有，则执行步骤h)，如否，则执行下一步；四种情况为：当前周期的机器状态空间中的任一条指令和所选指令间不存在数据相关；数据依赖图中任一指令与所选指令的弧上延迟值为0；数据依赖图中任一指令到所选指令的弧上的循环迭代数差值不为0；模调度中任一指令所在的级数不为所选指令所在的级数；

g) 置 feasible 为 false，然后执行步骤 k)，其中，feasible 为判断是否存在周期内的依赖关系的逻辑变量；

h) 将指令 op 加入当前周期的测试空间中，置 feasible 为 TRUE，调用模板匹配函数为当前周期的测试空间状态中的指令寻找模板，并实现有限状态自动机的状态转移；

i) 判断有限状态自动机的状态转移是否成功？如果是，执行下一步，如果否，执行步骤 l)；

j) 测试成功，把当前周期的测试空间的值赋给当前周期的机器状态空间，更新所选指令的绝对槽值，从指令集中删除所选指令，然后执行步骤 b)；

k) 在当前周期的机器状态空间中为所选指令做上已选标记，然后执行步骤 c)；

1）微调度结束。

在上述技术方案中，所述指令集为模调度认为放在同一 cycle 中发送的指令集。

在上述技术方案中，步骤 a)中所述每条指令的级数值为所述指令在扁平调度中的位置除以启动间距。

在上述技术方案中，所述绝对槽值表示所述方法结束后指令的绝对槽值。

由上可知，本发明所述的一种支持有向有环图的微调度方法，在运用重排（Reorder）技术和协调（Negotiate）技术编排模调度认为能在同一 cycle 中发射的指令集合的时候，除了考虑指令间的依赖关系以外，还要同时考虑指令间弧上的级数值和指令所在的级数，实现对“回边”的支持；避免软件流水模调度中出现的分拆问题（Split issue），减小了出现指令 cache 访问不命中（I-Cache miss）的可能性，提高了并行编译效率，从而进一步提高了编译优化性能。

附图说明

图 1a 是本发明实施例中循环体的伪代码示意图；

图 1b 是本发明实施例中循环体循环展开后的前三个循环示意图；

图 1c 是本发明实施例中一个迭代内的数据依赖图；

图 1d 是本发明实施例中循环的执行调度情况；

图 2a 是本发明实施例中一个迭代内扁平调度（II=2）的示意图；

图 2b 是本发明实施例中流水线执行示意图；

图 3a 是本发明实施例中循环迭代差值非 0 时寄存器分配前后示意图；

图 3b 是本发明实施例中指令级数不等时寄存器分配前后示意图；

图 4 是本发明实施例中支持有向有环图的微调度方法流程图。

图面说明：

图 1d 中，latency = 1，omega = 1，II = 1；I_1到 I_7表分别表示第 1 个 cycle 到第 7 个 cycle。

图 3 中，TN 表示临时变量（Temporary Name）；TN 表示全局临时变量（Global Temporary Name）。

图 4 中，流程图中字母含义：ops_list 为模调度认为可放在同一 cycle 中发送的指令集；

Stage 为模调度中指令所在的级数（op_{stage} = op 在扁平调度中的位置(location)/启动间距(II)）；

NULL 表示空集；

prev_state,cur_state 分别表示前一周期和当前周期的机器状态空间，temp_state 表示当前周期的机器状态的测试空间，若测试成功，则把 temp_state 的值赋给 cur_state;.

tried 为一布尔变量，其值为真表示 cur_state 中该 op 已被选过一次；

绝对 slot 值表示 MSMDDG 算法结束后指令的绝对槽值；

op 表示从 ops_list 中选出的一条指令；

inst 是 cur_state 中的任意一条指令；

Arc_latency (inst, op)表示 DDG 中指令 inst 到指令 op 的弧上的延迟值；

Omega (inst, op) 表示 DDG 中指令 inst 到指令 op 的依赖关系所跨越的循环迭代数。

Stage (inst)表示模调度中指令 inst 所在的级数（即 location/II），该值已在程序的一开始就计算完毕；

feasible 为判断是否存在周期内的依赖关系的逻辑变量；

Bundle_Helper 为微调度中现有模板匹配函数，其功能是在当前周期内，试图通过有限状态自动机（FSA: Finite State Automata）的状态转移，为 temp_state 状态空间中的指令寻找合适的模板。

具体实施方式

上文提到，模调度属于 Move-then-scheduling 技术，是跨越循环回边(back-edge)一条一条的移动指令，它构建的“数据依赖图”（DDG: Data Dependence Graph）不同于传统的“有向无环图”（DAG: Directed Acyclic Graph），而是一种“有向有环图”（DCG: Directed Cyclic Graph）。

它不仅包含循环内同一迭代（iteration）指令间的依赖关系，而且还包含不同迭代间的循环携带依赖关系（loop-carried dependence）。正是后者，通过循环回边（如果从 DDG 的初始节点到节点 a 的每条路经都要经过节点 b，则说节点 b 支配节点 a（b dom a）。如果 b dom a，则边 a→b 称为“*回边*”。如图 1c 中节点 1 指向自身的回边），把传统的 DAG 变成了“有向有环图”【Vicki H. Allan, Reese B. Jones, Randall M. Lee, Stephen J. Allan. Software pipelining. ACM Computing Surveys (CSUR). 1995,9(27) : 376∽432】。我们把这个“有向有环图”记为 DDG（N，A），其中，N 是表示所有指令的节点的集合，A 是表示所有依赖关系的弧的集合。每条弧都用一个序偶（omega, latency）标记。迭代距离（omega）表示依赖关系所跨越的循环迭代数，延迟（latency）表示第一条指令产生的结果能为第二条指令所使用而必需消耗的时间。

为了说明 ORC 中微调度如何对 DDG 做出支持，如图 1 所示，图 1a 中示出的循环体经过三次循环展开变成了图 1b，我们看到不仅同一迭代内

有依赖关系，不同迭代间也有依赖关系。这些依赖关系可以用图 1c 的 DDG 来充分表示。节点 1 指向节点 2 弧上的序偶(0, 1)表示同一迭代内 latency = 1 的依赖关系；节点 1 指向自身的回边上的序偶为(1, 1)，意味着当前迭代（iteration）的第一条指令和下一迭代的第一条指令也有 latency = 1 依赖关系。由于模调度在计算启动间距（II）时会保证 II * omega >= latency，所以当模调度选定 II 调度完毕后，便已经考虑了所有的依赖关系，包括迭代内和迭代间的依赖关系。

换句话说，一方面，模调度在进行“扁平调度（Flat schedule：即对一个 iteration 的调度）”时，要考虑一个迭代中相继的若干指令之间的依赖性（如图 2a 中示出的指令 2,3 被扁平调度调到同一 cycle，证明他们之间没有依赖关系）。另一方面，由于数据依赖图是有环的，模调度在调度一条指令时，还必须要考虑这条指令和扁平调度中所有“模 II 余数相同”的指令之间的依赖性，因为他们是同时执行的。如图 2b 中 I_1，I_3，I_5中的指令是同时执行的（I_i：i mod 2 = 1； i 为指令在扁平调度中的位置 location），I_2，I_4，I_6中的指令是同时执行的（I_i：i mod 2 = 0）。方框内的指令是流水线稳定时一个循环中的指令，称为一个级数（stage），它的长度等于 II。一条指令所在的 stage = location / II。Cycle I_5中的指令 6 和指令 1 分别来自于 stage2 和 stage0。

模调度结束之后，接下来由微调度负责 bundling。为了支持有向有环图，微调度必须要考虑 Omega 和 Stage 两项因素。

如图 2b 所示，Cycle I_5中的指令 6,2,3,1 在扁平调度中的位置模 II 余数相同，它们在 EPIC 结构中是在同一 cycle 中一起发送的。假设指令 6 和指令 2 之间有先读后写（RAW: Read After Write）的依赖关系（见下例），按照常见的调度方法，我们是不能把它们放到同一个 cycle 之内的，因为后一个 adds 要使用前一个 adds 的结果，必须要等待一个 cycle（latency = 1）。但从有向有环图可以得知，它们之间的弧上的权值里 omega = 1，也即指令 2 是来自下一个 iteration 里的。对这种 omega!=0 的情况，SWP 在 bundling 之后的寄存器分配（Register Allocation）可以轻易的通过寄存器重命名消除这一 RAW 的相关性，参见【B. R. Rau, M. Lee, P. P. Tirumalai, M. S. Schlansker. Register Allocation for Software Pipelined Loops. Proc. of the SIGPLAN '92 Conf. on Programming Language Design and Implementation. 1992,6 : 283∽299】。事实上 ORC 原有的软件流水部分也是这么做的。所以，当 omega >= 1 时，我们实际上是可以把这两条指令放到同一 cycle 的，图 3a 示出了指令 6 和指令 2 在软件流水做 Register Allocation 前后的情况。

Cycle I_5中指令 2 和指令 3 在 Flat Scheduling 中就被放到了同一个 cycle，证明其间没有相关性。指令 6 和指令 1 还可能是下面这种情形。同样，指令 6 到指令 1 也是先读后写（RAW）的依赖关系，如图 3b 所示，st1 的源操作数要使用 adds 的结果操作数。按常见的调度方法，我们也不能把它们放到同一个 cycle 中。但从模调度提供的信息可得知，指令 6 和指令 1 不是位于同一个 stage。前者来之 stage2，后者来自 stage0。

对这种情况，SWP 也会在 bundling 之后的 Register Allocation 中通过寄存器重命名消除它们之间的相关性，参见【B. R. Rau, M. Lee, P. P. Tirumalai, M. S. Schlansker. Register Allocation for Software Pipelined Loops. Proc. of the SIGPLAN '92 Conf. on Programming Language Design and Implementation. 1992,6 : 283～299】。所以，当它们的 stage 不相同的时候，实际上也可以放进同一个 cycle 中。图 3b 示出了指令 6 和指令 1 在软件流水做 Register Allocation 前后的情况。

综上所述，为了实现对“回边”的支持，微调度在运用重排（Reorder）技术和协调（Negotiate）技术编排模调度认为能在同一 cycle 中发射的指令集合的时候，除了考虑指令间的依赖关系以外，还要同时考虑指令间弧上的 omega 值和指令所在的 stage。这样才能和 bundling 之后 Register Allocation 配合起来，取得较好的优化性能。

下面参见图 4 详细说明本发明实施例中支持有向有环图的微调度方法，如图 4 所示，支持有向有环图的微调度方法包括以下步骤：

步骤 101，计算指令集（ops_list）中每条指令的级数值（Stage）；其中，stage = location / II；ops_list 为模调度认为可放在同一 cycle 中发送的指令集，location 为在扁平调度中的位置，II 为启动间距；

步骤 103，判断指令集（ops_list）是否为空？如果是，执行步骤 123；如果否，执行下一步；

步骤 105，判断机器当前状态空间（cur_state）是否已满或所有指

令（op）均有 tried 标志，即指令是否已被选过？若是，执行下一步，若否，执行步骤 109；

步骤 107，完成前一周期的机器状态空间 prev_state 的模板指派，更新 prev_state 中指令的绝对 slot 值，把当前周期的机器状态空间值赋给前一周期的状态空间，把当前机器的状态空间置空，即 prev_state = cur_state，cur_state = NULL；

步骤 109，从指令集 ops_list 中选一指令 OP，为其分配功能部件，把当前机器的状态空间 cur_state 赋给当前周期的测试空间 temp_state；

步骤 111， 根据数据依赖图，检查当前机器的状态空间的每一条指令 inst 与指令 op 的相关性。即通过判断是否有下列四种情况之一来判断是否有相关性，如有，则执行步骤 115，如否，则执行下一步；

在步骤 111 中，四种情况为：当前周期的机器状态空间中的任一条指令 inst 和该指令 op 间不存在数据相关；数据依赖图中指令 inst 与指令 op 的弧上延迟值为 0，即 Arc_latency (inst, op) = = 0；数据依赖图中指令 inst 到指令 op 的弧上的 Omega 不为 0，即 Omega(inst, op) != 0；模调度中指令 inst 所在的级数不为指令 op 所在的级数，即 Stage (inst) != Stage (op);

步骤 113，置 feasible 为 false，然后执行步骤 121，其中，feasible

为判断是否存在周期内的依赖关系的逻辑变量；

步骤 115，将指令 op 加入当前周期的测试空间 temp_state 状态空间中，置 feasible 为 TRUE，调用 Bundle_Helper 为 temp_state 中的指令寻找模板，并实现 FSA 的状态转移，其中，Bundle_Helper 为微调度中现有模板匹配函数，其功能是在当前周期内，试图通过有限状态自动机（FSA: Finite State Automata）的状态转移，为 temp_state 状态空间中的指令寻找合适的模板；

步骤 117，判断 FSA 状态转移是否成功？如果是，执行下一步，如果否，执行步骤 121；

步骤 119，测试成功，把 temp_state 的值赋给 cur_state，即 cur_state = temp_state，更新指令 op 的绝对 slot 值，从指令集 ops_list 中删除该指令 op，然后执行步骤 103；

步骤 121，在 cur_state 中为指令 op 做上 tried 标记，然后执行步骤 105；

步骤 123，微调度结束。

综上所述，本发明为了实现对“回边”的支持，微调度在运用重排（Reorder）技术和协调（Negotiate）技术编排模调度认为能在同一 cycle 中发射的指令集合的时候，除了考虑指令间的依赖关系以外，还要同时考虑指令间弧上的 omega 值和指令所在的 stage。这样才能和 bundling

之后 Register Allocation 配合起来，取得较好的优化性能，避免软件流水模调度中出现的分拆问题（Split issue），减小了出现指令 cache 访问不命中（I-Cache miss）的可能性，提高了并行编译效率，从而进一步提高了编译优化性能。

```
for (i=l; i<=n; i++)
O1: a[i + l] = a[i] + 2
O2: b[i] = a[i + l] * 3
O3: c[i] = b[i] - l
O4: d[i] = c[i]
```

图 1a

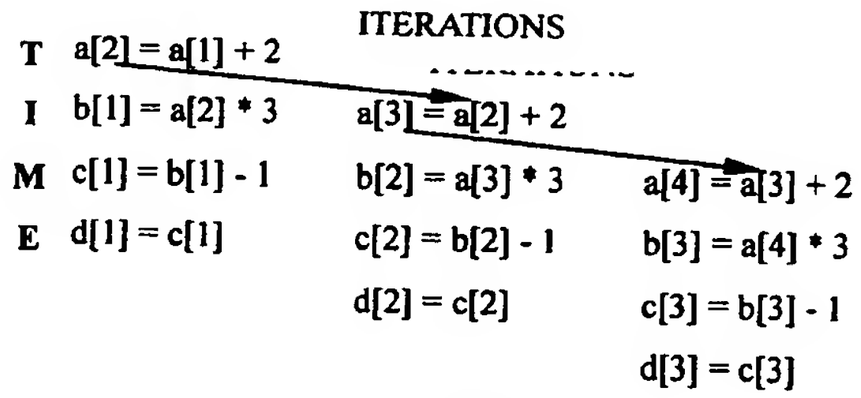


图 1b

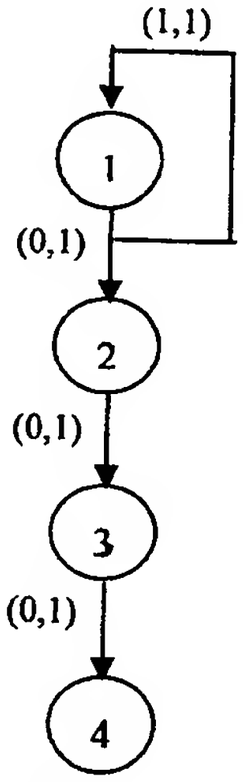


图 1c

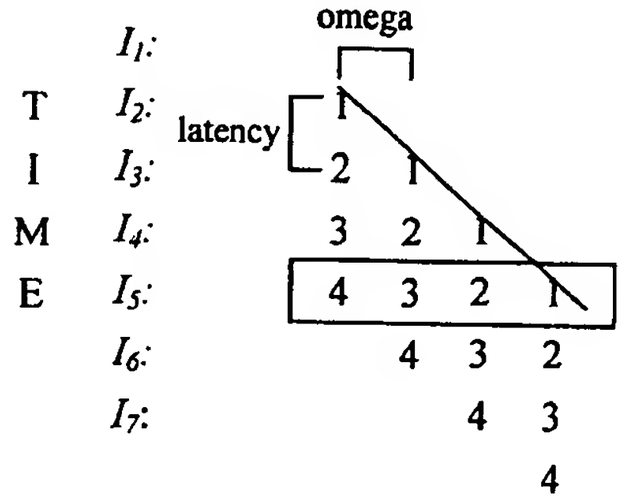


图 1d

F_1: 1
F_2:
F_3: 2,3
F_4: 4,5
F_5: 6
F_6: 7

图 2a

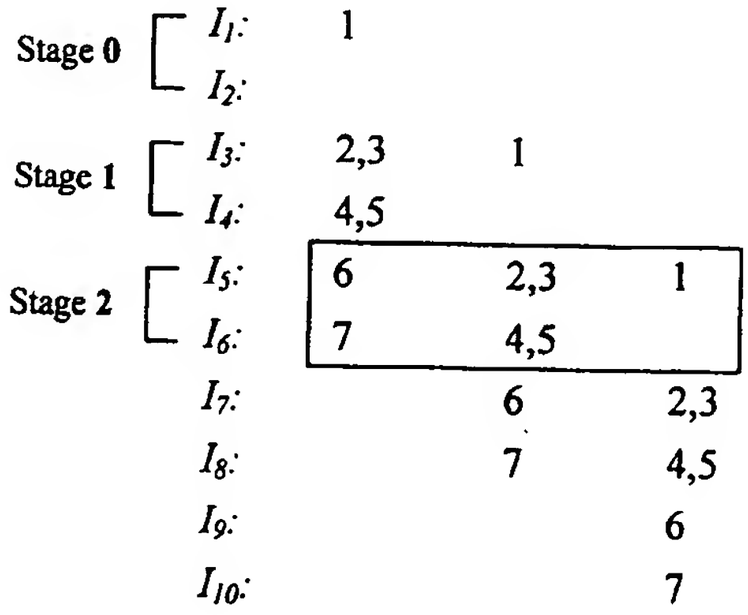


图 2b

```
[6] GTN1333 :- adds   -32   GTN1333[1];      //  GTN1333  = -32 + GTN1333
                      RAW
[2] TN1331  :- adds   -24   GTN1333[1] ;     //  TN1331   = -24 + GTN1333

 (.....After Register Allocation.....)
} { .mii
        ld1 r42 = [r14], -32                 // [stage1]
        adds r43 = -32, r44                  // [stage1]
        adds r52 = -24, r44                  // [stage1]
}
```

图 3a

```
[6] TN1343 :- adds    -8    GTN1349[1] ;          //  TN1343 = -8 + GTN1349
                      RAW
[1]           :- st1   [TN1343]   TN1304 ;       //  st[TN1343] = TN1304

 (.....After Register Allocation.....)
} { .mmi
        ld1 r47 = [r46]                      // [stage 1]
        ld1 r35 = [r57]                      // [stage 0]
        adds r52 = -8, r57                   // [stage 0]
 } { .mmi
        st1 [r59] = r41                      // [stage 1]
        st1 [r53] = r48                      // [stage 1]
        adds r41 = -4, r57 ;;                // [stage 0]
}
```

图 3b

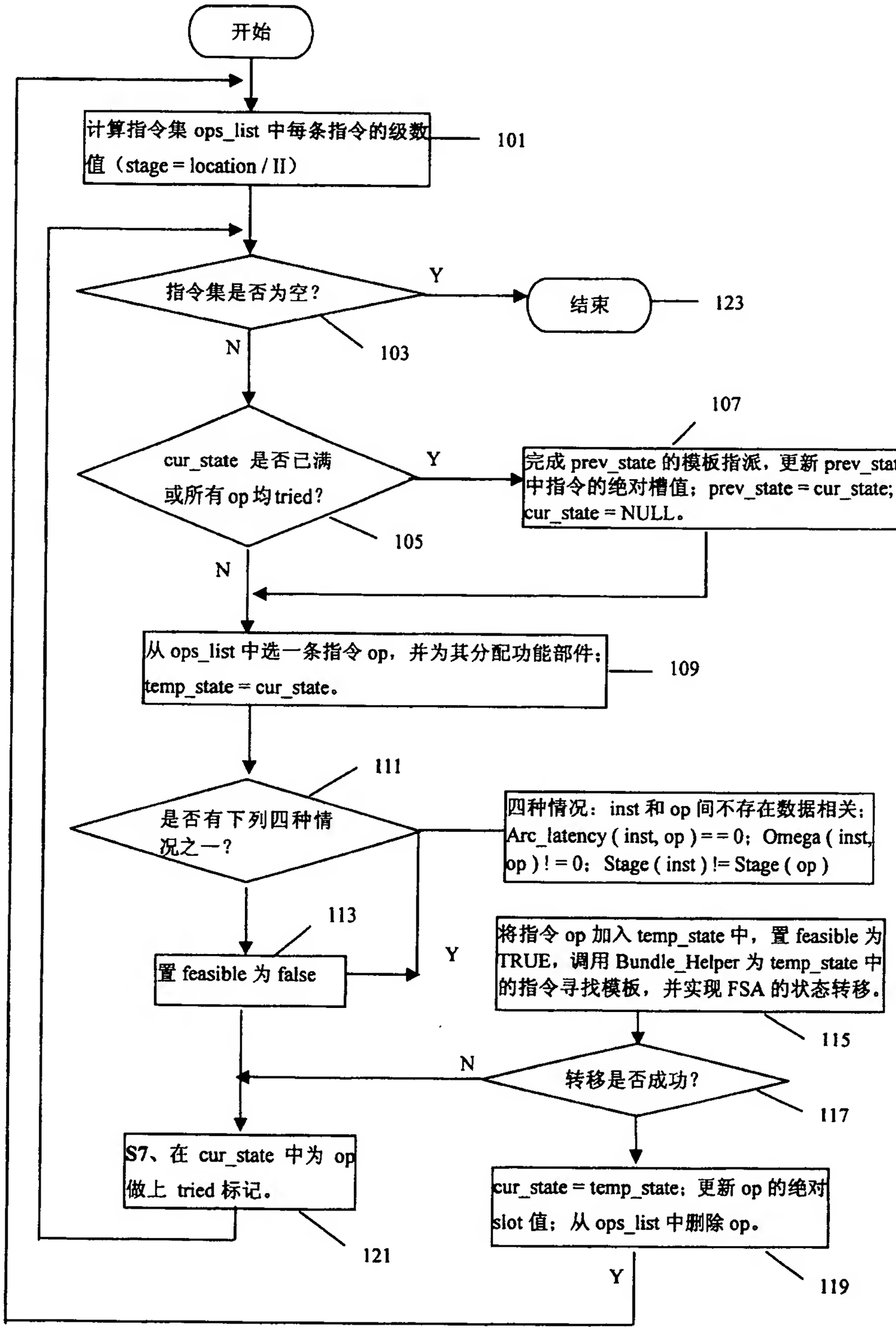

开始
计算指令集 ops_list 中每条指令的级数值（stage = location / II）
101
指令集是否为空？
Y
结束
123
N
103
cur_state 是否已满或所有 op 均 tried？
105
Y
107
完成 prev_state 的模板指派，更新 prev_state 中指令的绝对槽值；prev_state = cur_state; cur_state = NULL。
N
从 ops_list 中选一条指令 op，并为其分配功能部件；temp_state = cur_state。
109
是否有下列四种情况之一？
111
四种情况：inst 和 op 间不存在数据相关：Arc_latency (inst, op) = = 0；Omega (inst, op) ! = 0；Stage (inst) != Stage (op)
置 feasible 为 false
113
Y
将指令 op 加入 temp_state 中，置 feasible 为 TRUE，调用 Bundle_Helper 为 temp_state 中的指令寻找模板，并实现 FSA 的状态转移。
115
转移是否成功？
N
117
S7、在 cur_state 中为 op 做上 tried 标记。
121
cur_state = temp_state；更新 op 的绝对 slot 值；从 ops_list 中删除 op。
Y
119

图 4